NASA-TM-112244

016824

Sorting for Particle Flow Simulation on the Connection Machine

Leonardo Dagum[1]

Report RNR-90-017, October 17, 1990

National Aeronautics and
Space Administration

Ames Research Center
Moffett Field, California 94035

ARC 275 (Rev Feb 81)

Sorting for Particle Flow Simulation on the Connection Machine

Leonardo Dagum [1]

Abstract. This paper investigates the sorting requirements of a particle simulation and analyzes the sorting algorithms currently in use on sequential, vector, and data parallel implementations of particle flow simulations. Particle simulation requires sorting N integers in the range $[1, O(N)]$ and takes $O(N)$ running time on sequential or vector machines. The data parallel implementation of a particle simulation is shown to be non–optimal with running time $O(N \log N)$. Until recently, there have been no optimal parallel integer sorting algorithms. This paper presents an optimal deterministic algorithm for parallel sorting in a particle simulation. The algorithm first identifies ordered subsets within the disordered set of particles, and then merges these subsets in a novel fashion which takes $O(1)$ time with N processors and uses $O(1)$ additional memory per processor. The algorithm is optimal only when constrained to merging $O(1)$ ordered subsets and loses optimality when applied to sorting a fully disordered set.

1.1 Introduction

Of increasing interest to NASA and the fluid mechanics community in general has been the development of accurate and efficient methods for treating problems in rarefied flow. Renewed interest in the rarefied flow regime is a consequence of current efforts to design aerospace vehicles to operate in the upper atmosphere where rarefied flows are encountered. Problems in this regime are most commonly handled through the method of direct particle simulation, and in particular through use of the direct simulation Monte Carlo (DSMC) method (Bird (1976)). However, many of the algorithms of the DSMC method are not suited for "single instruction–stream, multiple data–stream" (SIMD) execution and therefore are incompatible with the architectures of current supercomputers. Much effort has been spent in developing new particle simulation algorithms more compatible with supercomputer architectures, and this effort has lead to the development of the Stanford particle simulation (SPS) method (Baganoff and McDonald (1990)). In addition to removing the data dependencies which would inhibit SIMD execution, the SPS method reduces the

[1] The author is an employee of Computer Sciences Corporation, Mail-Stop T045-1, NASA Ames Research Center, Moffett Field, CA 94035. The work reported here was performed while in the Department of Aeronautics and Astronautics at Stanford University. This research was supported in part by the National Aeronautics and Space Administration (NASA) under grant NAGW–965 and grant NCA2–313, and through DARPA by Cooperative Agreement NCC2–387 between NASA and the Universities Space Research Administration (USRA).

operation count per particle thereby significantly improving overall performance. However, both the DSMC and the SPS methods require some amount of sorting which, on the Connection Machine, has been implemented with a non–optimal algorithm (Dagum (1989)). This paper investigates the sorting requirements of a particle simulation and presents a parallel sorting algorithm explicitly for particle simulation with optimal performance.

1.1.1 Why Sort?

The selection of collision candidates in a particle simulation requires identifying particles occupying the same volume in physical space and from these selecting a representative sample to collide. For this purpose, physical space is discretized by a grid of cells and only particles occupying the same cell are possible candidates for collision. Each cell is identified by a cell index and a particle is associated with a single cell. In general, a sorting operation is required after the particles are moved such that particles occupying the same cell can be easily accessed.

A fundamental difference between the DSMC method and the SPS method is in the rule used to select colliding particles in the simulation. The amount of sorting required in the simulation is dependent on the selection rule, therefore it is necessary to understand these differences. Baganoff and McDonald (1990) classify selection rules according to the role given to the sample of candidate pairs taken from a cell. In the SPS method the sample size is not specified by the selection rule, therefore one has the freedom of using whatever algorithm is most natural for sampling candidate pairs from a cell. Because of the greater freedom allowed in the sampling of candidate pairs, this selection rule can be implemented with greater ease on different computer architectures. For this reason it is termed the *natural sample size* selection rule.

In the DSMC method, the sample size is explicitly specified for each cell and therefore is coupled to the the selection rule. Consequently, there is very little freedom in the choice of algorithm for sampling candidate pairs from a cell. Typically the particles must be fully sorted by order of their cell so that the correct number of pairs can be sampled from each cell. Baganoff and McDonald (1990) refer to this selection rule as the *constrained probability* selection rule because it ensures that probabilities of selection do not exceed unity for any sampled pair.

1.1.2 About This Paper

Dagum (1989) describes a data parallel implementation of the SPS method where each particle in the simulation is mapped to a single processing element in the architecture. This implementation uses a parallel radix sorting algorithm to order

the particles on every time step. The primary purpose of this paper is to present an alternative *optimal* parallel sorting algorithm which can be used either in the context of a particle simulation as described in Dagum (1989) or as a parallel merge algorithm when the number of lists to be merged is $O(1)$.

A parallel algorithm is optimal if the product of its processor complexity and its parallel time complexity is lower bounded by the minimum sequential time to solve the problem. In other words, the *processor* $\times$ *time* complexity is the same as the minimum sequential complexity of an algorithm. The sequential complexity of general sorting, i.e., where the range of the keys is unbounded, is known to be $O(N \log N)$ (cf. Knuth (1973)). However, when the range of the keys is finite, sorting does not have to be based on comparison of keys and the sequential complexity then is $O(N)$. This class of sorting is often referred to as *integer sorting* (cf. Aho *et al.*) because the keys either are, or can be mapped to, integers in the range $[1, O(N)]$.

In a particle simulation there are a finite number of cells for particles to occupy. Therefore only integer sorting is required and the optimal running time of a particle simulation is $O(N)$ where N is the number of particles. Section 2 of this paper describes the sequential and vector integer sorting algorithms used in both the DSMC and the SPS methods. Section 3 describes the parallel radix sorting algorithm used in the data parallel particle simulation and shows this algorithm is not optimal for integer sorting. Section 4 presents a new, merge based data parallel sorting algorithm explicitly for particle simulation which has optimal performance.

1.2 Sorting for Particle Simulations on Sequential or Vector Machines

The first sort which will be discussed here is the *bucket sort* (cf. Aho, Hopcroft, and Ullman (1974), or see *distribution counting*, Knuth (1973)) used in the DSMC method. In the DSMC method the particle indices are sorted by order of the particle's cell index (i.e. the index identifying the cell that a particle occupies). The sorted list is stored in a separate array called the *cross-reference array* (Bird (1976)). To identify the particles in a cell it is only necessary to know the cell density (i.e. the number of particles in the cell) and the first index in the cross-reference array of the particles in the cell. The latter will be referred to as the *starting index* of a cell.

The bucket sort employs a separate array with an element (or "bucket") for every possible key value. In one pass through the disordered set of keys, the number of

occurrences of each key is summed and stored in the key's bucket. Next, in a pass through the buckets, the running sum of the values in the buckets is computed and stored as the starting indices for the keys. Finally, in another pass through the disordered set of keys, the occurrence of each key is added to the starting index for the key to create the *rank* of that key. By moving each key to the position given by its rank, the set becomes ordered. In a particle simulation, the first step above corresponds to computing the cell density and the second step corresponds to computing the starting index for each cell.

The bucket sort cannot be fully vectorized although Boyd (1991) shows how it can be partially vectorized. Figure 1.1 is a schematic of the sorting process as performed by Boyd and will be described here in the context of a particle simulation. The unvectorizable elements of this algorithm are the calculation of the cell densities and the counting of previous *cell occurrences* in the table of particles. The term "cell occurrence" is used to mean, for a particular particle, how many other particles before it in the table occupy the same cell. In other words it is an enumeration of the particles in a cell. If this enumeration is available in a separate array then the last step of the algorithm, which is the creation of a rank for each particle, can be vectorized. The algorithm proceeds as follows. First, in one pass through the table of particles, the particles in each cell are enumerated and the final count for each cell becomes the cell density. In the figure, the state of the cell density array is shown for every step of the enumeration. The next step in the algorithm involves computing an array of starting indices for each cell simply by carrying out the running sum of the cell densities. Finally, the rank of each particle is computed by adding a particle's enumeration to the starting index for its cell. The first step of this algorithm cannot be vectorized because it is impossible to ensure that two particles in a vector do not also occupy the same cell. The second step cannot be vectorized because the running sum calculation is inherently data dependent.

In the SPS method the number of collision candidate pairs is left unspecified. The number of pairs sampled from a cell will depend on the implementation; this number is used by the selection rule to adjust the probability of selection accordingly. Typically one creates $n/2$ pairs in a cell where n is the cell density. Therefore on a sequential machine there is no need to order the particles before creating pairs, one can simply go directly to creating a known number of pairs of collision candidates to be used in the collision routine. This is accomplished by going through the table of particles once and keeping track of every occurrence of a cell in a separate array. The separate array is a mapping in memory of the cells in physical space. It is referred to as the *space map* (McDonald (1990)) and is analogous to the array of buckets used in the bucket sort. However, unlike the buckets, the space map is

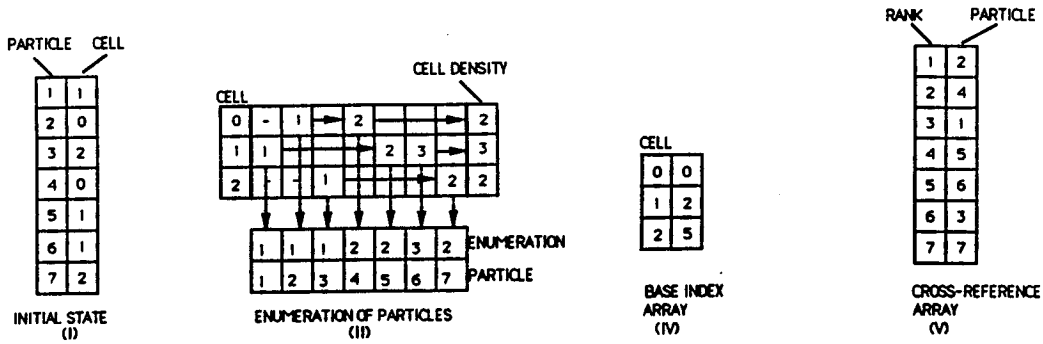

Figure 1.1
Schematic of steps in the DSMC sort algorithm

used to store either the index of a particle or a zero.

Figure 1.2 is a schematic of the collision candidate pairing algorithm used in the vectorized SPS method. One performs a single pass through the table of particles, checking the appropriate element in the space map for another unpaired particle in the same cell. If this is the case one creates a pair and zeroes the element in the space map. Otherwise the particle's index is stored in the space map as an unpaired particle. In figure 1.2 the state of the space map is shown for each step in the pairing process so it is possible to see how consideration of each particle in the table changes the state of the space map. Paired particle indices are stored at the time they are created in two arrays which are later used in the collision routine.

In the strictest sense the collision candidate pairing algorithm cannot be fully vectorized because it is impossible to ensure that two or more particles being processed in a vector do not occupy the same cell. Such a situation has two possible outcomes depending on the current state of the space map. If at the time the vector is being processed there are no unpaired particles in the cell such that the corresponding entry in the space map is zero, then only one of these particles will have their index written into the space map and the other one will essentially be ignored as a candidate collision partner. This is not an error since the number of pairs created in a cell is counted and used in computing the probability of selection. However, if at the time the vector is processed there does exist an unpaired particle in the cell with its index stored in the space map, then this particle will get paired with both particles in the vector. This can lead to an implementation–dependent outcome for the collision of these particles. The collisions are carried out in a vectorized fashion, therefore if the same particle collides twice in a vector the outcome

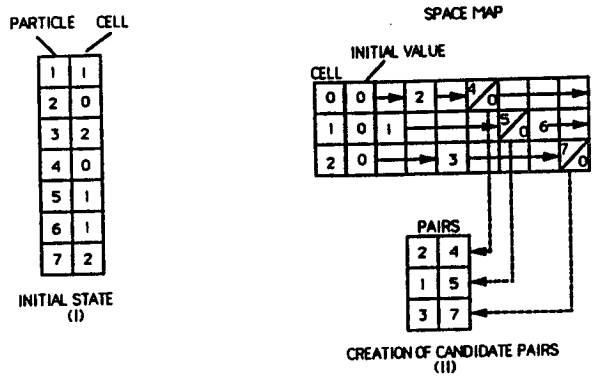

Figure 1.2
Schematic of collision candidate pairing algorithm used in the vectorised Stanford particle simulation method. The end result is a list of candidate pairs of colliding particles.

is unpredictable and very likely not to conserve momentum and energy.

As indicated by McDonald (1990), the probability of introducing an error through the data dependency described above is very low. Not only is it necessary for two particles from the same cell to have been in the same vector during the pairing process, it is also necessary for the two pairs to be accepted for collision before an error is introduced. Furthermore, since the effect of this error is small there is an acceptable tradeoff in accuracy for performance.

It is possible in some degree to quantify the heuristic argument given above by deriving the probability of two or more particles in a vector occupying the same cell. To do this we must make the following assumptions:

(i) the cell populations are all identical.

(ii) particle indices are independent of the particle locations, therefore the indices can be thought of as randomly distributed amongst the particles.

The second of these assumptions is generally true, however the first is true only in implementations of the DSMC method where cell volumes have been chosen to give a uniform cell population throughout the flow domain. For the cubic cells of unit width employed with the SPS method it is not reasonable to expect uniform cell populations, nonetheless one can proceed assuming (i) is true and then see how the result is affected when (i) does not hold.

Given the two assumptions above, one can find for any particle in the flow field the probability p that it occupies a particular cell c_j $(j = 1, \ldots, C_{tot})$ as

$$p = 1/C_{tot} \tag{1.1}$$

where C_{tot} is the total number of cells in the flow field. Conversely, the probability q that this particle *not* occupy a particular cell c_j is given by

$$q = 1 - 1/C_{tot}. \tag{1.2}$$

Now consider a group of N_{vec} particles, where N_{vec} is the number of elements in a vector (64 for the Cray 2). The probability of k particles in the group occupying the same cell is given by the binomial distribution

$$f(k) = \binom{N_{vec}}{k} p^k q^{N_{vec}-k} \tag{1.3}$$

where $\binom{N_{vec}}{k}$ are the binomial coefficients defined as

$$\binom{N_{vec}}{k} = \frac{N!}{k!(N_{vec}-k)!}. \tag{1.4}$$

The distribution is normalized so $\sum_{k=1}^{N_{vec}} f(k) = 1$. To determine P, the probability of two or more particles in a vector occupying the same cell, one must evaluate

$$\begin{aligned} P &= \sum_{k=2}^{N_{vec}} f(k) \\ &= 1 - f(0) - f(1) \\ &= 1 - q^{N_{vec}-1}(q + N_{vec}p). \end{aligned} \tag{1.5}$$

Substituting $q = 1 - p$ gives

$$P = 1 - (1-p)^{N_{vec}-1}(1 + (N_{vec}-1)p). \tag{1.6}$$

Typically the number of cells C_{tot} is large so $p \ll 1$ and

$$(1-p)^{N_{vec}-1} \approx 1 - (N_{vec}-1)p \tag{1.7}$$

and equation 1.6 simplifies to

$$\begin{aligned} P &\approx (N_{vec}-1)^2 p^2 \\ &\approx \frac{(N_{vec}-1)^2}{C_{tot}^2}. \end{aligned} \tag{1.8}$$

In a moderate size simulation one may employ 10^6 particles and about 5×10^4 cells. The probability of finding two particles in the same cell in a vector of 64 particles is then $P = 1.6 \times 10^{-6}$. One pass through the table of particles requires about 16000 vectors, therefore over 1000 time steps one would employ 1.6×10^7 vectors and one could expect to find $1.6 \times 10^7 \times P \cong 25$ vectors with two or more particles occupying the same cell.

The analysis above assumes that all cells have the same population therefore all particles are equally likely to occupy a particular cell. When the cell populations are not uniform, randomly chosen particles are more likely to be occupying the cells with greater population. Conversely, the chosen particles are less likely to be occupying cells with less population. Therefore one can remove from consideration those cells with less population and employ the result in equation 1.8 but with C_{tot} adjusted to reflect the greater weight given to the more populated cells. In the typical simulation of the previous paragraph there were an average of 20 particles per cell. In the actual flow one would probably find some cells with no particles and some with 150 or more. As a worst case scenario, assume an average of 100 particles per cell is significant, therefore with 10^6 particles there are $C'_{tot} = 10^3$ significant cells in the flow. Now over 1000 time steps there are 625 vectors which have two or more particles occupying the same cell. As a worst case this is still an insignificant error considering that there are a million particles in the flow and the calculation has been carried out over a thousand time steps.

It is of interest now to investigate what effect there would be in the DSMC method if the sorting algorithm there ignored the problem of two or more particles in a vector occupying the same cell. For simplicity let us assume that in sorting the particles there is exactly one vector with two particles occupying the same cell, c_j. First consider the effect on the calculation of the cell density. The cell density is the final result of enumerating the occurrences of a cell in the table of particles. If two particles in a vector occupy the same cell, then in processing that vector there will be two copies of the current cell count, each of which will be incremented by one. As a result the computed cell density for cell c_j will be less than the actual value by one. In the next step one computes the running sum of the cell densities

to get the starting index for each cell. The error in the cell density of c_j propogates to the starting indices of all cells indexed above c_j, and all those starting indices are less than the actual value by one.

Now consider the effect on the enumeration of the cell occurrences. The two particles in the same vector occupying cell c_j both receive the same enumeration. Therefore those two particles receive an equal rank. Furthermore, since the computed starting indices are less than the actual values by one, the ranks computed for any particles occupying cells indexed above c_j will be off by one. The cross–reference array is created by moving the particle indices to their sorted positions. In the cross–reference array one array element, CR_i, will receive two different values from the same vector but only one will be written. All array elements above CR_i (i.e. elements $CR_{i+k}, k = 1, 2, \ldots, N - i - 1$) will receive the particle index which should have gone to the next array element (i.e. to element $CR_{i+k+1}, k = 1, 2, \ldots, N - i - 1$). The last array element, CR_N, will be left unwritten to. Nonetheless, since the cross–reference array is accessed by the starting indices, and since the starting indices for particles ranked above CR_i are off by one, the cross–reference array still will provide the correct particle indices. In other words, the cross–reference array still will correctly identify the particles occupying the same cell and the only error will be in incorrectly computing the cell density for cell c_j. The argument above can be extended to cases where more than two particles in the same vector occupy the same cell or more than one vector has multiple particles which occupy the same cell. In all cases the cross–reference array will provide the correct particle indices and the only error will be in the calculation of the cell density. In conclusion, the partially vectorized bucket sorting algorithm described by Boyd (1991) could be fully vectorized with the introduction of negligible error.

On the Connection Machine it is not practical to employ either the DSMC sorting algorithm or the SPS candidate pairing algorithm. As discussed above, some error is introduced if two or more particles in a vector also occupy the same cell, and equation 1.8 gives the probability of such an event occurring. On the Connection Machine the "vector length" is effectively as great as the active virtual processor (VP) set. Since each virtual processor represents one particle, on the Connection Machine $N_{vec} = N$ and one will always find two or more particles in a "vector" occupying the same cell. This is a situation where the relatively small vector length of the Cray 2 allows what is in the strictest sense a scalar algorithm to be carried out in a SIMD fashion. One could emulate a smaller vector length, N'_{vec}, on the Connection Machine by looping through all the processors in the VP set with only N'_{vec} processors active at a time. However, the poor performance of individual pro-

cessors makes such a process very costly, and simply from a load balancing point of view it would be very inefficient regardless of the speed of individual processors. Consequently, on the Connection Machine one can virtually rule out as unfeasible any algorithm which requires almost scalar behavior over a large data set. Fortunately, alternative algorithms can be used and these are the topic of the remainder of this paper.

1.3 Radix Sort

The Connection Machine instruction set (PARIS) includes an instruction for finding the rank for each element in a disordered set of data (see Thinking Machines (1989)). This is the `CM_rank` instruction which, in version 5.x of the Connection Machine software, is simply a radix sorting algorithm written in microcode and optimized for performance. Hillis and Steele (1986) describe this algorithm for the Connection Machine. Sorting N elements with a maximum value of C_{tot} (the greatest value of a cell index in the current context) requires $\log C_{tot}$ passes through the N elements of the data set. Each pass considers a single bit of the sort key (the cell index in this case) beginning with the least significant bit and proceeding to the most significant bit. The elements with a zero bit are enumerated first, and the elements with a one bit are enumerated above these. Therefore if there are c elements with a zero bit they get assigned distinct integers y_k ranging from 1 to c. The remaining $N-c$ elements with a one bit then get assigned distinct integers y_k ranging from $c+1$ to N. The values y_k are then used to permute the elements such that all the elements with a zero bit precede the elements with a one bit. By proceeding through the $\log C_{tot}$ bits of the sort key the set gets ordered.

Figure 1.3 is a schematic for this algorithm. Beginning with the same disordered data set of figures 1.1 and 1.2, two pairs of enumerations are carried out to sort the set. In this example the maximum key value is 2, therefore only two bits are required to represent all the key values. The first pair of enumerations is used to re–order the particle indices based on the value of the least significant bit in the cell index. The new order is shown in the figure and is labelled "rank[1]" to indicate it is the ranking after examining the first bit of the key. The second pair of enumerations uses the rank[1] values in re–ordering the indices and thus arrives at the rank[2] result which here is the final result since there are only two bits in the key.

On the Connection Machine each re–ordering of the data set requires a general router communications event in the form of a `CM_send` (this corresponds to a con-

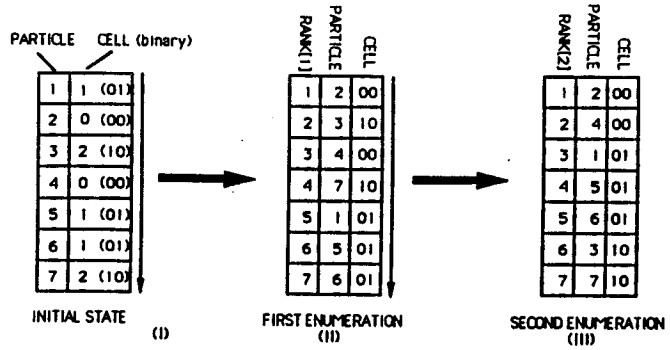

Figure 1.3
Schematic of radix sort algorithm for maximum key size of two bits.

current write in a parallel random access memory (PRAM) model of computation), therefore for the radix sort, as presented, there must be two enumerations and one "send" per bit of the key. This ratio of enumerations to sends can be changed simply by examining more than one bit before re–ordering. In other words, one can look at two bits of the key and carry out four distinct enumerations before re–ordering, or one can look at three bits and carry out eight enumerations before re–ordering and so forth. In general 2^j enumerations are required to re–order j bits. Therefore for every j bits in the key there will be 2^j enumerations and one send. On the Connection Machine enumerations are about 14 times faster than sends and the cost of the radix sort is proportional to $(2^j + 14)/j$. This function is minimized for $j = 3$, in other words re–ordering 3 bits in the key at a time. It is interesting and perhaps surprising that the `CM_rank` instruction uses $j = 2$ and it is possible to write a faster ranking routine in PARIS by using $j = 3$. Finally, it is worth noting that bucket sorting can be thought of as a special case of a radix sort which lets j be the maximum number of bits in the key and, because the range of the key is known precisely, allows the 2^j enumerations to be performed in a single pass through the data set.

A final concern for the sorting algorithm is to maintain statistical independence between samples of collision candidate pairs taken from a cell over succeeding time steps. As is described in Dagum (1989), collision candidate pairs are identified on an even/odd basis, therefore the sorting algorithm must allow the order of particles within a cell to change if statistical independence of the sample of pairs is to be maintained. With the radix sorting algorithm a simple mechanism for accomplish-

ing this is to concatenate a random sequence of bits to the least significant bit of the key, and then sort on this expanded key. A fixed number of bits are concatenated but their values are random, therefore sorting on the expanded key will restore the order of particles between cells while changing the relative ordering within cells. Unfortunately, expanding the key in this manner introduces additional bits to sort and there is a corresponding penalty in performance.

Assuming the "scan model" of computation (Blelloch (1987)) for the Connection Machine (that is, the Exclusive Read Exclusive Write (EREW) model but including scan operations as unit–time primitives) the time complexity, T, for this algorithm using N processors is $O(\log C'_{tot})$ where $C'_{tot} = 2^k C_{tot}$ is the size of the expanded key when k bits are concatenated to the cell index. The processor bound, P, is $O(N)$ and the algorithm has a performance bound of $PT = O(N \log C'_{tot})$. If this sort algorithm is used in a particle simulation the computational time will scale linearly with the number of particles only if the total number of cells is held fixed. In practice, one tends to design a simulation with a number of cells which is proportional to the number of particles to be used, so $C_{tot} = O(N)$. Therefore, this performance bound is nonlinear. The parallel radix sort is not optimal for integer sorting and is not recommended for a particle simulation other than for its simplicity and robustness.

1.4 Sorting by Merging Ordered Subsets

This section describes a very fast parallel ranking algorithm most suited for a two dimensional particle simulation. The algorithm proceeds by identifying ordered subsets in the full set of particles. The bulk of the work involves the merging of these ordered subsets into a single ordered set. The algorithm has a performance bound of $O(N)$ therefore it is optimal for sorting in a particle simulation.

1.4.1 Two Fundamental Observations

There are two fundamental observations which can be made about the dynamics of a particle simulation and which can be used to design an efficient ranking algorithm for this problem.

(1) On every time step the particles begin and end in an ordered state. The disordering of the particles occurs through their motion from one cell to another. Furthermore, the nature of this motion is such that on one time step only about a third of the particles will change cells, therefore the set is never greatly out of order. In fact it is precisely for this reason that there is statistical dependence between

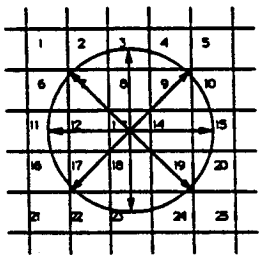

Figure 1.4
The maximum radius of motion over one time step is to a very high probability less than two cell widths.

even/odd pairings in succeeding time steps unless an effort is made to enhance the disorder (as discussed in the previous section).

(2) The motion of the particles is such that to a very high probability if a particle moves out of its current cell it will move only into one of its two immediate neighboring cells in the direction of motion, that is, particles do not move more than two cell widths per time step (see figure 1.4).

1.4.2 The Merged Ordered Subsets Sorting Algorithm

The merged ordered subsets sorting algorithm proceeds in the following manner. Making use of the first observation, at the beginning of the time step the particles are ordered and every processor is storing a value for its particle's current cell index. The particles then go through their motion after which a new value for the cell index must be computed. Both the old and the new values are stored, and now use is made of the second observation. It is convenient at this point to map the cell index into two dimensions and designate the pre–motion values by i, j and post–motion values by i', j'. Referring to figure 1.4 and assuming the second observation holds true, then it is obvious that a particle beginning in cell i, j has at most 25 different *and mutually exclusive* possibilities for its new cell location i', j'. (In three dimensions there are at most 125 mutually exclusive possibilities.) Conversely, if at the end of its motion a particle is occupying cell i', j', there are at most 25 mutually exclusive possibilities for its previous cell position i, j. Therefore one can divide the set of particles into 25 distinct and ordered subsets based on the 25 distinct possibilities for a previous cell location. In other words, because a particle in cell i', j' has 25 mutually exclusive possibilities for its previous cell location i, j, and because the particles were ordered in their previous cells, it follows that the order

must be preserved in 25 mutually exclusive subsets. The problem thus has been reduced to one of identifying these 25 ordered subsets and merging them into just one set.

Identifying each subset is accomplished by simply comparing the previous cell position to the current one. Also at this time it is convenient to count the number of particles in each of the subsets. This is useful later for optimizing the merging step since often there are less than 25 active sources in a time step.

To merge the subsets it is necessary to identify the lowest numbered processor for every cell in each subset, and then enumerate in each subset the processors representing a cell (see figure 1.5). A one dimensional grid, referred to as the "merging grid" and distinct from the physical grid of the simulation, is created with size great enough to contain an element for all the cells in the non–zero subsets. Therefore, if there are C_{tot} cells in the simulation and amongst all the cells N_s subsets are identified as active sources for the particles, then the merging grid must have at least $N_s C_{tot}$ elements. Note that N_s is usually less than 25 and once steady state has been reached it almost always is equal to 9. In other words, active sources usually include just the cell itself and its 8 immediate neighbors. Since C_{tot} is usually a power of 2, and since VP sets are restricted to powers of two, the greatest advantage occurs when N_s is 16 or less. If N_s is greater than 16 then the merging grid must have size $32C_{tot}$, but if N_s is 16 or less, then the merging grid will have size $16C_{tot}$. Since the merging grid then is half the maximum size, any operations performed with the processors of this merging grid will require about half the time required in the larger sized grid.

The primary task for the merging grid is to compute the "global ending index" for every active cell in each active subset. This is just the greatest rank for the particles in a particular cell and subset. For this purpose the `CM_send_with_add` instruction is used to determine the number density for every cell in each subset, then the `CM_scan_with_add` instruction is used to create a running sum of the number density which is then the global ending index. Therefore the merging grid now stores the greatest rank in the merged list for the particles in the cell it handles. The particles in each subset can be ranked by subtracting their enumeration in their cell and subset from the global ending index supplied by the merging grid.

The grid result is obtained by the particle processors (i.e. the processors representing the particles) through the use of the `CM_get` instruction. In order to minimize router contention it is necessary to minimize the number of particle processors active for this step. There are at most $N_s C_{tot}$ global ending indices, therefore at most $N_s C_{tot}$ particle processors need to get a global ending index from the merging grid. That is, only one particle processor for each cell and subset needs to get a

value from the merging grid. This value can then be copied across the rest of the particle processors in the cell and subset using the `CM_scan_with_copy` instruction. The lowest numbered processor for every cell in each subset was identified earlier in the algorithm and is used for this purpose.

Figure 1.5 is a schematic for the patterns of communication. Steps in the algorithm proceed from left to right across the page. In the first step the N_s active subsets are identified and the particles in each subset are enumerated with the enumeration re–starting at every cell. This requires N_s distinct pairs of scan operations, a pair for each set. The first scan is necessary to identify cell boundaries in a set and the second scan enumerates the particles in each cell. The next step of the algorithm requires all processors to send to the merging grid to create the cell number density. The global ending index is then created using a single `CM_scan_with_add`. Next, one processor in every cell in every subset gets its global ending index from the merging grid. This is depicted in the figure by an arrow with heads on both ends to emphasize that this operation requires communication in both directions. Finally, this value is copied across the processors in the cell in each subset by using N_s distinct `CM_scan_with_copy` operations. Now the processors can compute their rank simply by subtracting their enumeration within the cell (step 1 of figure 1.5) from the global ending index computed in the merging grid.

1.4.3 Maintaining Statistical Independence

It was claimed above that maintaining statistical independence of pairings between time steps is a concern of the simulation. In using the radix sort it was necessary to concatenate random bits to the end of the key and order the particles on this expanded key. The merged ordered subsets algorithm maintains elements of randomization in two ways. The first of these comes about from the manner in which the N_s subsets are mapped to the merging grid; the second is a result of employing the merging grid to compute the global ending index as opposed to the global starting index for a cell and subset. It is shown in this section that these mechanisms for randomization are not sufficient and it is necessary to further enhance the randomization.

It is worthwhile at this point to analyze the requirements of statistical independence. More specifically, we would like to be able to answer the question "how many identical pairs can one expect between two successive time steps if at each time step the choice of candidate collision pairs is made independent of the previous time step?" This would allow us to gauge quantitatively whether a particular algorithm maintains statistical independence or not. For this purpose, consider an arbitrary cell and let n be the number of particles within it. One can create

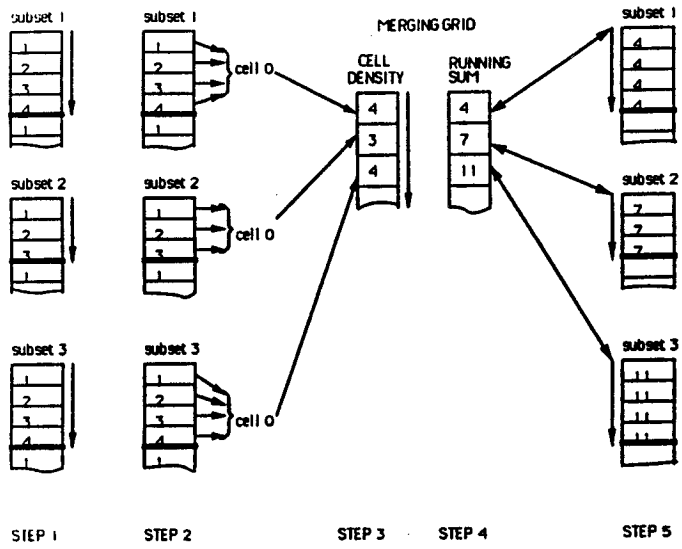

Figure 1.5
Schematic of the communications pattern in the merged ordered sets sorting algorithm.

$N = \binom{n}{2}$ different pairs from these particles. In a simulation one actually creates only k pairs where typically $k = n/2$. Clearly there are $\binom{N}{k}$ different ways to choose k pairs from N. The probability of making any particular selection must be $\binom{N}{k}^{-1}$. Now consider the probability that in making two selections no two pairs are found common. Since the first selection takes k pairs out of the pool of available combinations, in the second selection only $N-k$ pairs are available. So there are $\binom{N-k}{k}$ ways of choosing k pairs without using any of the same pairs chosen in the previous selection. Therefore the probability that in the second selection there are no pairs from the first selection must be

$$P(0) = \frac{\binom{N-k}{k}}{\binom{N}{k}}. \tag{1.9}$$

Now consider the probability that in making two selections exactly one pair is common in both. The first selection takes $N-(k-1)$ pairs out of the pool, and there are $\binom{k}{1}$ ways of choosing the one common pair from the k selected first. Therefore there are $\binom{N-(k-1)}{k}\binom{k}{1}$ ways of choosing k pairs in the second selection with one pair common to the first selection. The probability that the second selection has one pair common to the first must then be

$$P(1) = \frac{\binom{N-(k-1)}{k-1}\binom{k}{1}}{\binom{N}{k}}. \tag{1.10}$$

This can be generalized to a probability distribution for finding i common pairs in two selections as

$$P(i) = \frac{\binom{N-(k-i)}{k-i}\binom{k}{i}}{\binom{N}{k}} \qquad i = 0, 1, 2, \ldots, k. \tag{1.11}$$

This series is known as the *hypergeometric series* and its mean is given by (cf. Guttman, Wilks and Hunter)

$$\mu = \frac{k^2}{N}. \tag{1.12}$$

Substituting our values for k and N in terms of the cell density n, we find for large n the mean or expected value of the distribution goes to 1/2. This can be interpreted to mean that if one were to count, over samples taken from many cells with large cell densities, the number of pairs common over two time steps, the average value would be 1/2 if the samples were selected independently.

Now consider the merged ordered subsets algorithm and how randomization is introduced there. There are two immediately available mechanisms for randomization. The first is to use a random permutation of N_s for mapping the N_s subsets to the merging grid. This is not sufficient since the order of the particles is still preserved within each subset. The second is to reverse the order of enumeration of the particles in each cell and subset by using the global *ending* index in computing the rank. This also is insufficient since it is ineffective when the subset cell density is even. Recall that the purpose is to produce a different pairing of those particles which remain in a cell between subsequent time steps. Pairs are created by matching the even–addressed particles with the odd–addressed ones, therefore reversing the enumeration changes the pairing only when the number of particles in the enumeration is odd. These points are discussed more extensively in Dagum (1990), the significant outcome is that additional randomization is required from the algorithm before it can be used in a particle simulation.

Additional randomization must be applied at two scales. There must be randomization in the order of the particles in each subset, and there must be randomization of the ordering across the subsets within a cell. The opportunity for accomplishing the first of these exists in the enumeration stage of the algorithm. The order of enumeration within each subset can be shuffled in a deterministic fashion at very little cost. The shuffling is performed by re–numbering the processors in a cell after

the regular enumeration has been performed. Two shuffling algorithms are employed because each shuffling algorithm is deterministic and two applications of the same shuffle to a data set produces no change in the relative ordering. Alternating between two different shuffling algorithms ensures there is no correlation between samples taken two time steps apart. The diffusion and convection of the particles through the flow field is such that, for the time steps used in most practical applications, there is no correlation between the particles in a cell three time steps apart.

Shuffling requires the particle processors to know the subset cell density. This is obtained through the use of the `CM_scan_with_copy` instruction. Figure 1.6 has a schematic for each of the two shuffling algorithms. In the first shuffling algorithm, the processors which previously had an even number are re–numbered continuously from 1 to $n/2$ and the processors which previously had an odd number are re–numbered continuously from $n/2+1$ to n. On the next time step when these processors are paired as even–with–odd the renumbering will effectively make pairs as even–with–even and odd–with–odd in terms of the addresses in the previous time step. In the second shuffling algorithm, the numbering of the odd processors is left intact but that of the even processors is reversed. Therefore if n_{ev} is the number of even processors, then their numbering is changed from from $2, 4, 6, \ldots, 2n_{ev}$ to $2n_{ev}, 2n_{ev}-2, \ldots, 2$. On the next time step when these processors are paired as even–with–odd the renumbering will effectively make pairs as first–with–last, second–with–secondlast, and so on, in terms of the addresses in the previous time step. The reason for two shuffling algorithms should now be clear. If either of the two shuffles are applied twice in sequence, the resulting pairing is unchanged. However by alternating between the two shuffles it is possible to guarantee different pairings for at least two succeeding time steps. By the third time step most of the particles in a cell have left and it is not necessary to worry about correlations over more than two time steps.

Randomization of the order across subsets must now be addressed. This randomization is absolutely necessary because the particles within each subset are highly correlated amongst themselves. Recall that each subset is identified on the basis of the direction of motion of the particles. For example, one subset will include all the particles which arrived at a new cell from the neighboring cell directly below. Therefore all those particles will tend to have velocities in the upward direction regardless of how they are rearranged amongst themselves. Some correlation like this will exist in each of the subsets. To eliminate such correlations it is necessary to mix the order across subsets.

Of the two shuffles described above, only the second one will allow mixing across

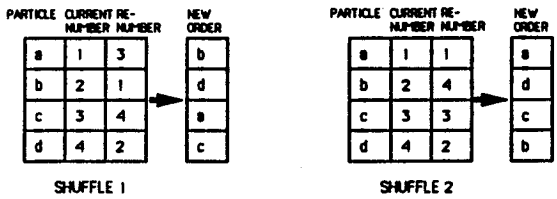

Figure 1.6
Two deterministic shuffling algorithms. These shuffles are applied on alternating time steps to the particles in each cell in order to ensure sufficient mixing.

subsets. The extension of this algorithm to shuffling across a complete cell is straightforward; n_{ev} becomes the number of even processors for the cell, as opposed to the number of even processors in a subset for the cell. The new numbering will result in pairings between particles of different subsets in the cell. In a similar fashion the algorithm can be applied separately to the first and last half of the processors for the cell. By alternating between shuffling over the complete cell and shuffling the first and last halves separately, the pairing of collision candidates is made random. The effectiveness of these shuffling algorithms is ascertained in Dagum (1990) through calculations for thermal relaxation in a heat bath and for shock wave profiles. Non–random samples of candidate collision pairs lead to incorrect collision frequencies in the flow and these become immediately evident in the results of such calculations.

1.4.4 When Assumptions Fail

The algorithm has been presented from a physical perspective and in the context of a generic cell in a generic time step. Two observations of the dynamics of the simulation were necessary for the algorithm to be valid. It is necessary now to discuss the situations where these observations do not hold true and the algorithm cannot be used.

The obvious situation for which the sort will fail occurs when particles move over more than two cells in one time step. The assumption that particles do not move more than two cell widths in one time step is true to a very high probability, however given the statistical nature of the simulation it is impossible to rule out the possibility of a particle not holding to this assumption. It is a simple matter to trap these instances *before* carrying out the sort thus allowing the opportunity to employ a fully deterministic algorithm (e.g. the radix sort) instead.

A more common situation is for the first observation to fail. The first observation

claimed that the particles go from an ordered to a disordered state through their motion from one cell to another. This is not true at the upstream boundary of the wind tunnel where new particles must be introduced to maintain a uniform free stream. However the introduction of new particles can be delayed an arbitrary number of time steps, therefore it is convenient to employ the radix sort on those time steps where new particles are introduced and use the merged ordered subsets sort on the other time steps. The alternative is to treat the introduction of new particles as an edge effect and handle it separately. Since the new particles can be introduced in an ordered fashion the algorithm can be modified to handle this situation by performing an additional merging. In other words, after ordering the particles in the flow one introduces the new particles and merges these with the flow particles. The merging now involves only two ordered subsets and proceeds much faster than with N_s subsets.

1.4.5 Performance and Extension to Three Dimensions

The performance of the merged ordered subsets algorithm depends to some degree on N_s, the number of subsets to be merged. In two dimensions, N_s is usually 9, and for this case the merged ordered subsets algorithm takes about 45% of the time of the radix algorithm on the Connection Machine (for a simulation using 2 million particles and 50,000 cells).

The performance of the algorithm scales linearly with the number of particles in the simulation. There is also some dependence of performance on C_{tot}, the number of cells, since this number determines the size of the merging grid to be used. However the fraction of work performed by the merging grid is decoupled from the fraction corresponding to the particles. Recall that the merging grid is used to perform one scan and one send operation. The time to perform these operations is dependent on C_{tot} and is independent of N, the number of particles. Therefore the algorithm can be characterized as scaling as $O(N + C_{tot}) = O(N)$, since $C_{tot} = O(N)$.

Perhaps most interesting from a theoretical point of view is the time complexity of the algorithm. The algorithm requires a constant number of send and scan operations to merge a constant number of subsets. We define the *extended scan model* of computation as simply the Exclusive Read Concurrent Write (ERCW) model but including scan operations as primitive unit–time operations. Note that we have extended Blelloch's scan model by allowing concurrent writes. This is realistic for the Connection Machine since the routing hardware is capable of combining multiple messages to the same address. We are *not* allowing concurrent reads as unit–time primitives since, on the Connection Machine, multiple reads from the

same address are buffered and handled iteratively. With the extended scan model, the time complexity, T, for N parallel processors is $O(1)$. The processor bound, P, is $O(N)$ therefore the performance bound is $PT = O(N)$, which is optimal for sorting over a fixed range of keys.

The distinguishing feature of this algorithm is that it is deterministic for the set of cases where it may be applied (see section 4.4). Furthermore, the cases where it cannot be applied can be determined *before* carrying out the steps in the algorithm. Note, however, that the algorithm is restricted to merging $O(1)$ ordered subsets. Because of this restriction, the algorithm uses only $O(1)$ additional memory per processor (i.e. the merging grid is of size $O(N)$), and takes $O(1)$ time to merge. An unrestricted version of this algorithm (i.e. where we allow $O(N)$ sorted subsets in the disordered array) would require $O(N)$ additional memory per processor and take $O(N)$ time to merge. Therefore it does not seem likely that this algorithm can be generalized to any integer sorting problem, nonetheless it is the only deterministic algorithm we know that is optimal for sorting in the context of a particle simulation. We should mention that Rajasekaran and Reif (1989) present an optimal *randomized* parallel algorithm for integer sorting. However, this algorithm is unsuitable for implementation on the Connection Machine because it requires allocating a *variable* amount of memory in each processor. Since the Connection Machine is SIMD, memory must be allocated uniformly in all processors and the memory requirements for this algorithm become excessive (the worst case would require $O(N^2/\log N)$ total additional memory). Furthermore, since the algorithm divides N records over $N/\log N$ processors, it would require rearrangement of the data mapping in the particle simulation to make use of this algorithm.

The algorithm has been presented and discussed for a two dimensional flow simulation. The extension to three dimensions is straightforward but does involve a loss in performance due to increased communications. In three dimensions there are at most 125 mutually exclusive sets instead of 25. Again one can expect for the vast majority of time steps only immediate neighbors will act as sources of particles in a cell, therefore in three dimensions 27 ordered subsets will usually be identified as opposed to 9 ordered subsets in two dimensions. The algorithm requires three scan operations with the particle processors per subset. In two dimensions these operations account for 55% of the time to rank. In the worst case, in three dimensions that fraction of the algorithm would triple in time so overall the new ranking algorithm would take about 2.1 times longer than for two dimensions. In addition, the merging grid would be twice as large in three dimensions and the time spent by the merging grid would double. However, the performance bound is still $O(N)$ therefore the algorithm remains optimal in three dimensions although the constant

factor is significantly greater than in two dimensions.

1.5 Conclusions

A necessary requirement of a particle simulation is that it have a minimum performance bound of $O(N)$ for N particles. This can be achieved on sequential or vector architectures by using integer sorting algorithms with complexity $O(N)$. However, these algorithms do not map well to data parallel architectures nor have there been any data parallel integer sorting algorithms with $O(N)$ performance bound presented in the literature. This paper analyzes the sorting algorithms currently in use for particle simulation with sequential and vector architectures, and presents the first deterministic and optimal integer sorting algorithm for particle simulation on a data parallel architecture.

Acknowledgment

I would like to thank Professor Donald Baganoff for his invaluable help in the development of the data parallel particle simulation.

Bibliography

[1] AHO, A.V., HOPCROFT, J.E., and ULLMAN, J.D., *The Design and Analysis of Computer Algorithms*, Addison Wesley Publishing Company, (1974).

[2] BAGANOFF, D. and MCDONALD, J.D., *A Collision-Selection Rule for a Particle Simulation Method Suited to Vector Computers*, Physics of Fluids A, Vol. 2, No. 7, pp. 1248–1259, (1990).

[3] BLELLOCH, G., *Scans as Primitive Parallel Operations*, Proceedings of 1987 International Conference on Parallel Processing, University Park, PA, (1987).

[4] BIRD, G.A., *Molecular Gas Dynamics*, Clarendon Press, Oxford, (1976).

[5] BOYD, I.D., *Vectorization of a Monte Carlo Simulation Scheme for Nonequilibrium Gas Dynamics*, to appear in Journal of Computational Physics, (1991).

[6] DAGUM, L., *Implementation of a Hypersonic Rarefied Flow Particle Simulation on the Connection Machine*, Proceedings of Supercomputing '89, Reno NV, (1989).

[7] DAGUM, L., *On the Suitability of the Connection Machine for Direct Particle Simulation*, Ph.D. Thesis, Department of Aeronautics and Astronautics, Stanford University, (1990).

[8] GUTTMAN, I., WILKS, S.S. and HUNTER, J.S. *Introductory Engineering Statistics*, John Wiley and Sons, Inc., Toronto, (1971).

[9] KNUTH, D.E., *The Art of Computer Programming: Vol. 3/Sorting and Searching*, Addison Wesley Publishing Company (1973).

[10] MCDONALD, J.D., *A Computationally Efficient Particle Simulation Method Suited to Vector Computer Architectures*, Ph.D. Thesis, Department of Aeronautics and Astronautics, Stanford University, (1990).

[11] RAJASEKARAN, S., and REIF, J.H., *Optimal and Sublogarithmic Time Randomized Parallel Sorting Algorithms*, SIAM Journal of Computing, Vol. 18, No. 3, pp. 594–607, (1989).

[12] THINKING MACHINES CORPORATION, The Connection Machine System: Parallel Instruction Set, Thinking Machines Corporation, Cambridge, MA, (1989).